Software-Managed Energy-Efficient Hybrid DRAM/NVM Main Memory

Software-Managed Energy-Efficient Hybrid DRAM/NVM Main Memory

Ahmad Hassan[1], Hans Vandierendonck[2], and Dimitrios S. Nikolopoulos[2]

[1]HANA Cloud Computing, SAP The Northern Ireland Science Park Queen's Road, Belfast, Northern Ireland BT3 9DT, United Kingdom
[2]The School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Northern Ireland BT7 1NN, United Kingdom

Abstract

This paper evaluates the viability of user-level software management of a hybrid DRAM/NVM main memory system. We propose an operating system (OS) and programming interface to place data from within the user application. We present a profiling tool to help programmers decide on the placement of application data in hybrid memory systems. Cycle-accurate simulation of modified applications confirms that our approach is more energy-efficient than state-of-the-art hardware or OS approaches at equivalent performance. Moreover, our results are validated on several candidate NVM technologies and a wide set of 14 benchmarks.

The key observation behind this work is that, for the workloads we evaluated, application objects are too short-lived to motivate migration. Utilizing this property significantly reduces the hardware complexity of hybrid memory systems.

1 Introduction

As DRAM scaling is potentially at an end, the industry is turning towards byte-addressable non-volatile memory (NVM) as a main memory replacement [4,8,23]. NVM technologies however have several drawbacks. The precise drawbacks differ between technologies. Generally speaking they include increased latency and dynamic energy for NVM accesses, reduced memory bandwidth and a fast wear-out of NVM devices compared to DRAM [4, 15, 25, 33]. On the plus side, NVM will scale to smaller feature sizes [4] and has around 100× lower static energy due to the absence of refresh operations [8].

Hybrid main memory, containing both DRAM and NVM has been proposed to address the issues underlying NVM [14, 15, 25, 26]. The idea is that a small amount of DRAM holds frequently accessed data and provides low latency and low dynamic energy. A large amount of NVM is used to implement a scalable memory system and store the majority of cold data. The key question to address is how to decide what data should be placed on what type of memory.

The state-of-the-art proposes various caching-based solutions that are operated by the hardware [15, 25] or the operating system [26]. These solutions try to second-guess the properties of the workloads and migrate large chunks of data, typically corresponding to the page granularity of the (virtual) memory system, between DRAM and NVM. This introduces run-time overhead and energy consumption due to the monitoring and the migration.

This paper proposes to manage the placement of data in a hybrid main memory at the fine granularity of application-level objects, which are individual program variables or memory allocations. Figure 1

[1]This is the postprint version of the paper appearing in the Computing Frontiers'15 conference with DOI 10.1145/2742854.2742886.

shows why operating on objects leads to higher energy savings. The dashed curve shows the cumulative distribution function of the difference in energy consumed for a 4 KB page when it is stored on DRAM in comparison to storing the page on RRAM. 61% of pages incur no extra energy, while around 8% of pages are clearly hot. This leaves 31% of pages in the grey zone. Likewise, the solid curve shows that 91% of objects are clearly cold and incur no added energy, while 9% of objects are clearly hot. This analysis shows that objects are a better granularity to decide placement in a hybrid memory hierarchy, as they are strongly biased towards one particular memory technology.

Pages are less effective for deciding placement as each page can hold a collection of various small objects, each with a distinct bias. As such, the page itself has no strong preference for a particular memory type. Moreover, objects are typically short-lived. As such, data migration is too slow to adapt to eventual changes in the access patterns of individual objects. Often, there is little need to migrate application-level objects.

In contrast, pages are alive throughout the execution of the program and, over time, they may hold objects with distinct properties. Page migration is necessary to deal with these fluctuations, but page migration comes at a cost: observing characteristics, learning patterns and reading and writing page-sized chunks of memory.

We apply our analysis in two distinct settings: embedded workloads executing on an embedded CPU, and more complex workloads executing on a high-end CPU. We find that there is little need to migrate application-level objects in our set of embedded workloads, which simplifies the design. Results are less pronounced for the complex workloads, where the effectiveness of object placement depends on the number and size of objects. To be precise, page migration performs better than object placement when hot objects are much larger than the page size.

We define a programming interface that allows the programmer to allocate memory either on DRAM or NVM. This requires support from the operating system and the system libraries (see Section 4). Moreover, we present a tool that helps programmers to analyze applications and to decide where key application objects should be allocated (Section 5). This tool profiles the application to collect memory access statistics and applies an analytical energy model to derive a placement for the objects. The programmer uses the output of the tool to encode the static placement of application objects on the appropriate memory.[1]

We demonstrate using cycle-accurate simulation that our technique outperforms state-of-the-art page migration techniques (Section 6). Software placement can provide up to 81.5% reduction of main memory energy while introducing no more than 3.9% slowdown. Moreover, we apply our placement tool to analyze various data granularities for managing hybrid memory systems and compare results across multiple NVM technologies. Our analysis shows that object placement performs consistently well across all investigated scenarios.

[1]One could also automate the process and encode placement information mechanically in the source code.

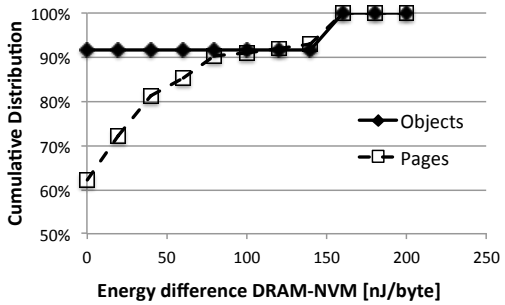


Figure 1: Histogram of DRAM energy minus NVM energy for objects and pages of the jpeg benchmark. Energy is normalized per byte.

Table 1: Latency (ns) and dynamic energy (nJ) for a 32-byte access and leakage power (mW/GB) for various memory technologies.

	Latency	Dyn. Energy	Leakage
	R/W, [ns]	R/W, [nJ]	[mW/GB]
DRAM	30.5/30.5 [22]	5.9/12.7 [22]	451 [22]
PCM	151.4/395.9 [9]	80.41/418.6 [9]	4.23 [8]
STT-RAM	105.0/77.21 [9]	16.60/21.05 [9]	4.23 [8]
RRAM	116/145 [9]	4.81/13.8 [9]	4.23 [8]

2 Background on NVM Technology

Emerging NVM memories like PCM, STT-RAM and RRAM enable memory chips that are byte-addressable, non-volatile, require near-zero leakage power and have density and latency closer to current DRAM technology. Unlike DRAM, NVM technology is characterized by asymmetric read, write, and erase latencies. Although none of these devices are in mass scale production yet, the anticipated device parameters such as access latency, dynamic energy and leakage power imply open research challenges in integrating such memories in the memory hierarchy along with DRAM.

PCM [27] exploits the properties of chalcogenide glass when heated and cooled at different speeds. It is the most mature technology among the NVM contenders. However, its latency and dynamic energy is worse compared to other technologies. In comparison to DRAM, PCM is 2× slower for array read and 43× slower for array write [15] while leakage power is 1% of DRAM [8].

STT-RAM [24] is a magnetic RAM device and stores data using the orientation of a magnetic field. STT-RAM has lower latency than PCM. The dynamic energy and density of STT-RAM is worse than DRAM. STT-RAM has better endurance (10^{15}cycles) compared to other NVM technologies [21, 24].

RRAM [24] uses measurable change in resistance due to the move of ions to represent 0 or 1. RRAM devices based on memristors are the most promising NVM technology for the realization of faster, high density, non-volatile and low-power main memory [11]. However, RRAM is challenging to manufacture at scale.

Table 1 presents parameter values that we obtained from the literature [8, 15, 22, 24] and/or through using NVSim [9]. NVSim simulates circuit-level performance, energy, and area models for PCM, STT-RAM and RRAM. We configured NVSim to jointly optimize dynamic energy and latency for a 1 GB memory chip with 32 byte word width. We used PCM cell model in 'current' mode for reading and writing, a 90 nm node, SET current of 75 μA and RESET current of 600 μA. For STT-RAM we used the MRAM cell at 65 nm node with RESET/SET current of 400 μA. For RRAM we used the memristor cell with default RESET/SET voltage of 0.8V and -0.8V respectively. PCM background energy is 1 % of DRAM [8]. As the background energy of STT-RAM and RRAM are not available in the literature, we assumed the same value.

The relative order of these numbers is in line with the literature in the field [4, 14, 15, 21, 25]. The conclusions of this paper apply regardless of the precise values for the device technologies, as long as DRAM and NVM are sufficiently different to warrant a hybrid memory.

3 Related Work

Research on hybrid main memory has predominantly pursued software-oblivious memory management executed by the hardware [14, 15, 25] or involving the OS [7, 23, 26]. In both cases, the memory access properties of large chunks of data are analyzed on-line and chunks are migrated to the most appropriate memory type. For practical reasons, the chunk size is chosen equal to the virtual memory page size, or in some cases to the internal row buffer size in DIMMs [32], which is of similar magnitude as the page

size.

While these solutions hide all complexity from the user software, they are all limited to operate on pages. It is the main contribution of this paper to show that pages are the wrong granularity of data to operate on for these purposes. Moreover, page-based techniques need to learn the read-write pattern and/or access frequency of pages, which incurs space, time and energy overhead.

Recent work has demonstrated that hardware caching of smaller DRAM chunks (e.g., 64-byte blocks) can be effective [13]. For this reason, we will also evaluate the effectiveness of smaller-sized chunks. However, the chunk size still bears no relation to the object size.

Software management of hybrid on-chip scratchpad memories extends existing techniques to differentiate between hot and cold on-chip data [12,19]. In this paper we study hybrid main memories, where accesses have been filtered by the on-chip caches. As such, different techniques are required.

Wang *et al* [31] consider the placement of data on a hybrid memory for a hard real-time multi-tasking system. Periodically arriving tasks must meet their deadline and may store their data on one memory type only. An optimal allocation through integer linear programming is computed. However, the requirement that all of a task's data must be stored on the same memory type is unnecessarily restrictive.

Closest to our work is the work of Li *et al* [18] who explore the use of NVM for scientific applications. They utilize several heuristics to place data on either type of memory, leading to 27% power savings. In contrast, this work uses analytical models to decide on data placement. Moreover, we develop the full software stack required to effect the data placement.

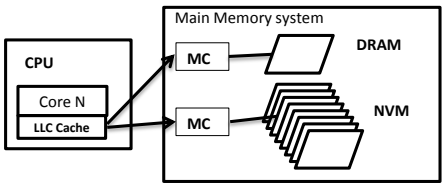


Figure 2: System architecture

4 Programming Interface and OS Support

The programming interface exposes the hybrid nature of the main memory (Figure 2) to the programmer. At the hardware level, the DRAM and NVM memory chips are assigned distinct physical address regions. This ties in neatly with how the BIOS reports DIMMs and their physical address ranges to the OS. In this case, the BIOS additionally reports the technology used to implement the DIMMs. The OS can then select to allocate virtual memory pages on either type of memory, depending on the requirements set by the programmer.

We extend the memory allocation methods to direct the allocation of memory on NVM or on DRAM. For `mmap` we add a new option to the flag argument. For the `malloc` family of functions we provide alternative functions prefixed with "`hotmem_`" (e.g., `hotmem_malloc`) which have the same arguments but they allocate memory on DRAM. The existing functions implement a default allocation on NVM.

We furthermore extend the linker file format to provide two versions of each type of data segment. For instance, for the ELF file format, the segment `.bss_hotmem` holds zero-initialized data that is frequently accessed in main memory, while `.bss` holds the cold data. The GCC annotation "`__attribute__((section("bss_hotmem")))`" effects global variable placement.

Stack data has a very high cache hit rate, often exceeding 99% [6,17]. As such main memory accesses are rare for stack data. If objects do attract a high number of repeating memory accesses, the programmer should explicitly allocate the data on the DRAM heap.

If migration is necessary, the programmer can allocate a new copy of the object on the opposite memory type and copy the data. If memory latency or bandwidth proves to be a bottleneck during migration, one may consider to speed it up with hardware support. However, we did not observe any

need for migration in our benchmarks due to the limited lifetime and memory access chararistics of objects.

For the OS, allocating memory on a hybrid memory system is similar to allocating memory in a non-uniform memory architecture (NUMA). Here, every NUMA region is further split in a DRAM region and an NVM region. As such, the OS and system libraries can utilize the same memory allocation algorithm for either type of memory. Moreover, virtual memory management and virtual address translation are the same for DRAM and NVM and are unmodified in comparison to a NUMA system.

OS implementations of NUMA contain various design decisions that may lead to violation of the programmer's allocation choice. E.g., Linux will not keep track of the NUMA preference of swapped-out pages and may swap them in in the "wrong" NUMA partition [3]. Moreover, there may be conflicting constraints when mapping pages into multiple virtual address spaces. These issues are common between NUMA and hybrid memory systems. Resolving them is out of the scope of this paper.

5 Object Placement Tool

The object placement tool derives a placement of objects, i.e., individual program variables and memory allocations, within the hybrid memory system with the objective of saving energy in the main memory system. The tool uses first-order analytical models of memory access time and energy to estimate the impact of placing an object on DRAM or NVM. The objective of analytical models is to maximize the energy savings through data placement on the hybrid memory system under a given performance degradation constraint λ, which is a user-configurable parameter. These properties are calculated for each object independently. Only the objects benefiting from DRAM most are placed on DRAM. This placement is subsequently encoded in the program source code. We use detailed simulation to validate the object placement derived from our tool.

5.1 Performance and Energy Model

The first-order performance effect of placing an object on NVM or DRAM is related to the memory access latency and bandwidth, which differs between NVM and DRAM. Our performance model takes into account only access latency [30]. Bandwidth issues can occur only when objects are frequently accessed, in which case we already select to place the object on DRAM due to the latency benefits.

The average memory access time (AMAT) incurred by memory accesses to one object o stored in a memory of technology τ is given by $AMAT_\tau(o)$:

$$AMAT_\tau(o) = \mu_r(o)L_{\tau,r} + (1 - \mu_r(o))L_{LLC} \tag{1}$$

where τ is either DRAM or NVM, $L_{\tau,r}$ is the latency to read a cache block in memory (see Table 1) and L_{LLC} is the latency to access the last-level cache (LLC). $\mu_r(o)$ is the number of read memory accesses made to o per load or store operation. Note that we are assuming a single-level cache, but the formula can be extended to multi-level caches.

Besides estimating performance, we also estimate the energy impact of storing an object in memory. We discern static energy, which is always present throughout the lifetime of an object and includes leakage and refresh energy, and dynamic energy, which is proportional to the frequency of memory accesses. Energy consumed by an object o is presented by the $AMAE_\tau(o)$ metric:

$$AMAE_\tau(o) = \mu_r(o)E_{\tau,r} + \mu_w(o)E_\tau + S(o)\ P_\tau T(o) \tag{2}$$

Here, $E_{\tau,r}$ and $E_{\tau,w}$ are the energy for reading and writing, respectively, a cache block to or from memory type τ. These parameters are computed from Table 1. The parameters $\mu_r(o)$ and $\mu_w(o)$ represent the

read access and write accesses to memory, respectively, as in the definition of AMAT. P_τ is the average leakage power per byte for memory type τ. The parameters $S(o)$ and $T(o)$ represent the size and lifetime, respectively, of the object o.

5.2 Object Placement

For any object o, we calculate the metric $\Delta AMAT(o) = AMAT_{DRAM}(o) - AMAT_{NVM}(o)$ to estimate the potential slowdown by placing the object on NVM. Similarly, we calculate $\Delta AMAE(o) = AMAE_{DRAM}(o) - AMAE_{NVM}(o)$ to estimate the energy gain by placing the object on NVM. The latter is typically a function of the trade-off between static and dynamic energy for the object.

We place objects such that energy is minimized and latency is raised by no more than a fixed percentage over a DRAM-only system. To this end, we sort the objects in order of increasing $\Delta AMAT(o)$ and place objects on DRAM in this order until DRAM is fully occupied. We partition the list of sorted objects $o_i, 1 \leq i \leq N$ by splitting the list at index s, such that objects $o_i, i \leq s$ are placed on DRAM and objects $o_i, i > s$ are placed on NVM. We determine the index s in order to meet the expected overall slowdown compared to a DRAM-only memory system:

$$\sum_{i=s+1}^{N} \Delta AMAT(o_s) \leq \lambda \sum_{i=1}^{N} AMAT_{dram}(o_s) \tag{3}$$

where λ is a user-configurable parameter. We set λ to 5% in this paper.

Note that this fairly simple algorithm works well because $\Delta AMAE(o)$ correlates with $\Delta AMAT(o)$. This happens because systems with small memory capacities tend to be dominated by dynamic energy [30]. As such, both quantities depend mostly on the number of memory accesses. If this correlation is absent, more complex partitioning algorithms may be used instead.

5.3 Tool Implementation

The object placement tool collects the required memory access statistics through a profiling run of the program. Hereto, it uses the LLVM compiler framework [20] to add instrumentation code to the analyzed program. The instrumentation code calls into a runtime library that collects statistics on the objects. The library calls include the declaration of statically allocated objects, the (de-)allocation of dynamically allocated stack and heap objects, and the interception of load and store assembly operations. The load and store operations are run through a simulator of the on-chip memory hierarchy in order to derive the frequency of cache misses and writeback events occurring for each object.

As the system libraries are not instrumented, calls to common system functions are intercepted at instrumentation time and the memory accesses performed by these functions are simulated on the memory hierarchy simulator [28]. Intercepted functions include string and memory copy and compare functions (e.g., `strcpy`) and file I/O (e.g., `read`). For some functions the (de-)allocation of objects is emulated as a side-effect (e.g., `fopen` and `fclose`).

6 Experimental Evaluation

We evaluate the effectiveness of software management of hybrid main memory using both the placement tool (Section 5.3) and cycle-accurate simulation.

6.1 Experimental Setup

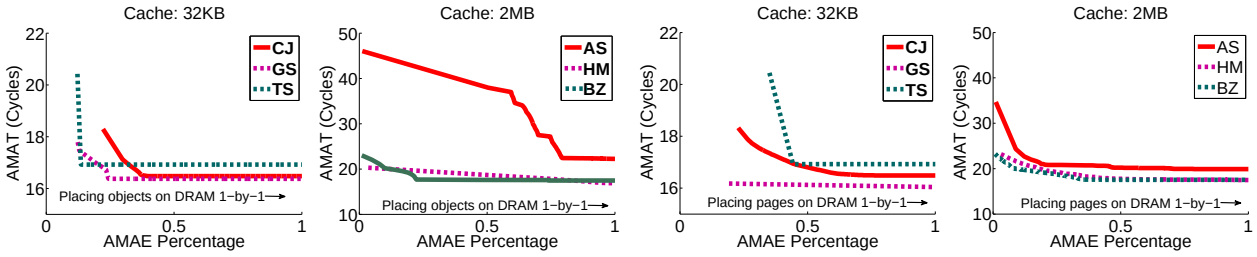
We used the GEM5 [2] cycle accurate simulator for the validation of data placement. For benchmarks in Table 2, we configured GEM5 with the ARM Cortex-A7 CPU model. The key property of this CPU

Table 2: Benchmarks for embedded CPU and Cache Configuration

Benchmarks	Suite	Description
CJPEG (CJ)	MediaBench [16]	JPEG encoding
Consumer Lame (LM)	MiBench [10]	MP3 encoder
SMG2000 (SMG)	ASCI Purple	Multigrid Solver
ESPRESSO (ESP)	SPEC'92	Expression optimization
Ghostscript (GS)	MiBench [10]	Language interpreter
Consumer Typeset (TS)	MiBench [10]	Typesetting tool
X264 (X264)	PARSEC [1]	H.264 video encoding

Table 3: Benchmarks for high-end CPU and Cache Configuration

Benchmark	Suite	Description
473.astar (AS)	SPEC CPU2006 [29]	2D path-finder
456.hmmer (HM)	SPEC CPU2006 [29]	Hidden Markov Models
401.bzip2 (BZ)	SPEC CPU2006 [29]	Compression / decompression
458.sjeng (SJ)	SPEC CPU2006 [29]	Chess game
462.libquantum (LQ)	SPEC CPU2006 [29]	Simulation of quantum computer
464.h264ref (HR)	SPEC CPU2006 [29]	Video compression
429.mcf (MCF)	SPEC CPU2006 [29]	Vehicle scheduling



(a) Object placement (b) Page placement

Figure 3: Placing data incrementally in the memory hierarchy and its impact on AMAT and AMAE. The left-most (right-most) point on each curve corresponds to NVM-only (DRAM-only) memory. Showing (a) object placement and (b) page placement.

for this work is its on-chip cache hierarchy, providing a 4-way associative 32 KB L_1 cache. For the set of benchmarks in Table 3, we simulated an X86 CPU. This system is configured with a two-level cache hierarchy, providing a 4-way associative 64 KB L_1 cache and a 4-way associative 2 MB L_2 cache.

For embedded system simulation, we configured the GEM5 with two memory controllers, one interfacing with 128 MB DRAM and one interfacing with 1 GB RRAM. For high-end system simulation, the GEM5 is configured with 256 MB DRAM and 1152 MB RRAM. The timing parameters for RRAM are derived from the Micron DDR3 datasheet [22] and prior literature [8, 15, 24]. Specifically, we derived the tRCD, tRP, tRRDpre and tRRDact parameters for RRAM following Lee *et al* [15]. We calculated tRCD as 2× larger than DRAM, tRP as 2.2× larger than DRAM, tRDDpre is 6 ns and tRDDact is 6.5 ns. Moreover, we modified the GEM5 memory controller to avoid write-back of clean cache blocks to memory. This not only improves the endurance of RRAM, but also saves write energy. Both DRAM and RRAM memory controllers use open page management policy with maximum 4 accesses per row before closing.

Finally, we extended GEM5 to expose the DRAM and NVM address ranges to the key memory allocation routine `mmap` (our `malloc` library builds on `mmap`) and to enforce the placement of global variables. We simulated fourteen benchmarks (Table 2 and 3). The benchmarks are simulated for a maximum of 2 G instructions in system emulation mode of GEM5. For the benchmarks in Table 2, we skipped the first 500 M instructions in the detailed simulation. For the benchmarks in Table 3, we skipped the first 2 G instructions in the detailed simulation.

We used the training dataset in all the benchmarks to train the model. We used the test dataset for GEM5 validation. The number of objects created by an application vary between the training and the

Table 4: Embedded CPU: Comparison of an NVM-only system (NVM), software placement (SWP) and RaPP.

	CPI Increase (%)			Energy Savings (%)		
Bench	**NVM**	**SWP**	**RaPP**	**NVM**	**SWP**	**RaPP**
CJ	18.44	2.90	5.37	-779.3	81.20	77.19
LM	91.53	1.80	14.10	-1375.80	69.13	73.70
SMG	98.55	3.90	11.60	-1779.10	23.44	66.50
ESP	26.36	3.24	15.00	-729.60	74.29	61.60
GS	47.83	2.36	12.30	-398.70	80.15	79.80
TS	41.02	2.70	19.60	-623.20	80.9	50.60

test dataset. In order to apply our model to a different dataset, we used the call site of each allocation to derive data placement. Our proposed model suggests the data placement for each call site on the basis of analytical models. The results in Table 4 and Table 5 show that our proposed model works well for different input datasets.

6.2 Object Placement

We apply the placement tool (Section 5.2) on the application objects and apply the same methodology on pages by handling each page as if it were an application object. Figure 3 plots for each object o_i in position i in the sorted list the cumulative values $\Sigma_{j=1}^{i}\Delta AMAE(o_j)$ (X-axis) and $\Sigma_{j=1}^{i}\Delta AMAT(o_j)$ (Y-axis). The results assume a 32 KB on-chip CPU cache and RRAM memory (see Table 1 for parameters).

From left to right, the curves show the transition in AMAT and AMAE when an object or a page is moved one-by-one from NVM to DRAM. The left-most point on the curve represents an NVM-only memory system as all objects are placed on NVM. The right-most point on the curve represents a DRAM-only memory system.

For all benchmarks, moving more objects onto DRAM results in an energy increase as the active amount of DRAM grows, but memory access latency shrinks as DRAM has lower latency. For some benchmarks, the optimal partitioning is well pronounced (e.g., typeset), while for others it is not (e.g., hmmer).

Comparing object placement (Figure 3a) against page placement (Figure 3b), we observe that page placement has limited energy savings potential as the curves show a knee at a higher percentage of energy savings. There are fundamental reasons for this. In many cases, pages are much larger than objects. As such, a page contains a collection of objects with possibly distinct preference towards DRAM or NVM. One may argue that objects may be placed in memory such that all objects on the same page have similar characteristics. Such techniques have been well studied, e.g., to minimize cache misses [5]. However, this does not negate our thesis that *software needs to be involved in placing objects* in order to minimize energy in a hybrid memory hierarchy.

It may appear that page-level placement and migration is superior to object placement when objects consist of multiple pages. Hardware or combined hardware/OS techniques have been studied to migrate pages as necessary. Such online techniques, however, incur significant overhead in terms of hardware real estate and its associated energy consumption. In this work, we observe that a large fraction of multi-page objects can be placed correctly from within the application, thus providing the same energy savings without the overhead. The next Section shows that object placement outperforms page migration.

6.3 Validation of Software Object Placement

We simulate software object placement on our GEM5 simulator and compare with the state-of-the-art Rank-based Page Placement (RaPP) policy [26]. RaPP uses a multi-level queue to identify frequently accessed pages. We follow the proposed design with 15 queues and implement a DRAM expiry time of 100μs.

Table 5: High-end CPU: Comparison of an NVM-only system (NVM), software placement (SWP) and RaPP.

	CPI Increase (%)			Energy Savings (%)		
Bench	**NVM**	**SWP**	**RaPP**	**NVM**	**SWP**	**RaPP**
473.astar	35.86	1.40	17.93	79.70	77.84	77.14
456.hmmer	50.79	2.66	15.54	68.30	71.47	63.00
401.bzip2	29.32	2.10	6.61	81.20	81.51	82.60
458.sjeng	28.67	2.45	17.00	74.32	79.82	74.90
462.libquan.	20.49	3.11	13.20	83.00	67.35	80.20
464.h264ref	11.84	2.16	11.84	76.30	76.88	78.20
429.mcf	47.60	0.61	21.20	38.20	70.66	49.20

Table 6: Embedded CPU: Energy savings through optimized data placement assuming a maximum slowdown of 5%. Results are shown for three granularities of data management and three NVM technologies.

	DRAM + RRAM (%)			DRAM + STT-RAM (%)			DRAM + PCM (%)		
Benchmark	**Block**	**Page**	**Object**	**Block**	**Page**	**Object**	**Block**	**Page**	**Object**
CJ	10.22	36.50	64.88	-40.79	24.54	55.05	-524.41	-228.79	32.69
LM	0.62	2.87	36.92	-34.38	-28.31	33.11	-466.63	-428.88	-95.86
SMG	1.36	2.22	11.29	-2.32	1.08	-2.38	-52.95	-46.06	-28.38
ESP	13.72	1.07	29.08	15.74	-9.46	22.45	-78.68	-99.48	-32.10
GS	7.68	56.44	85.37	-96.06	32.81	71.15	-1995.89	-535.00	-66.41
TS	7.73	65.41	81.45	-10.75	54.65	67.84	-282.26	-48.96	29.67
X264	10.50	62.25	76.10	-26.52	69.21	51.14	-419.60	-12.50	-44.68

Software object placement out-performs RaPP on both performance (CPI) and energy (Table 4) for all but one benchmark. Energy savings in the range 69% to 81% are realized while performance never degrades over 5%. In the case of SMG, the 5% maximum performance degradation set for object placement is not aggressive enough to enable the same energy savings as RaPP. In contrast, RaPP energy savings are in the range 50% to 80%. RaPP incurs slowdowns in excess of 10% for all but one benchmark.

Note that Table 4 does not include X264 due to issues with making its execution consistent across repeated simulations.

Table 5 shows the simulation results for the seven SPEC benchmarks. In general, the NVM-only system gives higher energy savings but CPI increases significantly, on average by 32 %. Software object placement provides a mechanism to limit the CPI increase to a desired value while maximizing energy savings. All measured CPI increases are within the 5 % threshold (Table 5). On the other hand, RaPP causes higher performance degradation in comparison to software object placement for all the benchmarks while it achieves less energy saving.

Table 7: High-end CPU: Energy savings through optimized data placement assuming a maximum slowdown of 5%. Results are shown for three granularities of data management and three NVM technologies.

	DRAM + RRAM (%)			DRAM + STT-RAM (%)			DRAM + PCM (%)		
Benchmark	**Block**	**Page**	**Object**	**Block**	**Page**	**Object**	**Block**	**Page**	**Object**
473.astar	93.83	79.44	34.43	93.54	79	27.37	93.50	78.09	26.61
456.hmmer	78.91	64.83	4.76	78.14	64.10	3.43	74.13	63.82	1.96
401.bzip2	67.51	66.31	77.70	66.29	65.14	76.33	64.11	63.82	71.84
458.sjeng	87.93	88.32	54.20	72.14	68.12	39.13	64.31	64.25	33.70
462.libquantum	69.03	41.64	4.71	67.24	40.14	3.22	64.33	39.14	2.89
464.h264ref	88.52	71.54	81.20	87.71	70.54	69.64	83.00	69.71	59.19
429.mcf	12.80	8.23	0.0047	12.08	6.10	0.0003	10.64	5.09	0.0003

Table 8: Percentage of objects larger than a page (4 KB) or block (64 B) and percentage of memory accesses to those objects for the high-end workloads.

	Object >block		Object >page	
Benchmark	**% Obj.**	**% Acc.**	**% Obj.**	**% Acc.**
473.astar	71.47	49.02	0.17	44.61
456.hmmer	27.30	99.65	0.02	97.53
401.bzip2	59.56	99.97	1.08	99.52
458.sjeng	35.02	64.40	10.20	42.56
462.libquantum	49.87	99.98	<0.01	99.98
464.h264ref	5.32	47.05	<0.01	4.18
429.mcf	12.50	99.98	4.16	99.96

6.4 Analysis of Placement Granularity

Tables 6 and 7 show the energy reduction for various NVM technologies and placement granularities while limiting potential slowdowns to less than 5% of a DRAM-only system. These energy savings are estimated using the profiling tool by measuring the lifetime of objects and the number of memory accesses. The lifetime of objects is measured at microsecond clock resolution. Lifetimes shorter than a microsecond are taken as 1 μs which results in a small error in the energy estimate. We consistently underestimate energy savings, however, as a more accurate clock will result in smaller lifetimes and thus less static energy wasted in DRAM in the baseline system. We will show shortly that the profiling tool is accurate and produces comparable results to the cycle-accurate simulator.

6.4.1 Embedded Scenario

Considering RRAM (Table 6, left), placement of objects has the potential for extremely large energy savings, over 64% for half of the benchmarks and as high as 85.3% of the main memory energy. These numbers correlate very well with the cycle-accurate simulation results (Table 4).

SMG is an outlier, only 11.29% savings, because memory accesses are evenly distributed among a large number of objects. As the dynamic energy is dominant in all of these benchmarks, there is hardly any gain in moving small objects on NVM. The energy savings due to NVM become more apparent when the static energy savings in NVM overcome the dynamic energy. The static energy depends on both the size and the lifetime of objects.

Page placement is less effective than object placement. The estimates (Table 6) are more negative than the simulation results which included page migration as well as page placement (Table 4). Nonetheless, we conclude that pages are not the right granularity for data placement and migration in the embedded scenario.

Block placement performs worse than page placement in our benchmarks. A closer look into the profiling trace reveals that there are many objects smaller than 32 bytes, so the fragmentation issue has not been adequately addressed. Moreover, many blocks get non-uniform accesses. As such the blocks that are placed on DRAM contribute to too much dynamic and static energy which cannot be overcome by the low static energy of NVM.

6.4.2 High-End Scenario

Results are different in the high-end scenario. This is mainly due to the difference in system characteristics. By utilizing a deeper memory hierarchy, more memory accesses are caught in the on-chip caches and never make it to main memory. Moreover, the workloads have higher complexity than the embedded workloads and have significantly larger memory footprints. Because of these effects, *the high-end scenario is dominated by static energy consumption, while the embedded scenario is balanced between static and dynamic energy.* This effect has been reported and analyzed in [30].

Considering the high-end scenario with RRAM (Table 7), placement of objects saves up to 81.2% energy. In this case, however, energy savings exceed 50% for just 3 benchmarks. We observe that the success of object placement depends strongly on object size. E.g., for h264ref, few objects are larger than a page (Table 8) and these objects attract few main memory accesses. As such, these large objects are uniformly cold and object placement works no worse than page placement. In contrast, in libquantum nearly all main memory accesses are targeted at a small set of objects larger than a page. It is not affordable to place any large object in its entirety on NVM due to the resulting slowdown. In contrast, page placement can exploit the imbalance of memory accesses made to each page of the large objects and place a subset of the pages on NVM, thereby saving more energy.

Note that the predicted energy savings for object placement (Table 7) are less than those measured in the simulator (Table 5). This is due to the setup: memory size is constant in the simulator (1.4 GB), while the profiling tool predicts energy savings relative to the actual size of the memory footprint. As such, energy savings predicted by the simulator are invariably high if the workload leaves a major part of memory unutilized.

Block placement outperforms page placement for all workloads, yielding energy savings up to 93.8% (Table 7). This analysis, however, shows the potential of block placement only as it includes no overhead (e.g., for locating the appropriate blocks). We conclude that block placement does promise significant additional energy savings in comparison to page placement. As such, it is worthwhile to explore efficient hardware techniques for implementing address translation at the block level [13].

6.5 Alternate NVM Technologies

Table 6 shows estimated energy savings for STT-RAM and PCM for the embedded scenario. A hybrid memory with STT-RAM often consumes more energy than a DRAM-only memory system when using page and block granularities for placement. The need for migration again shows up but it is stronger now then on RRAM, especially for block placement. Application objects, however, are providing similar energy efficiency on STT-RAM compared to RRAM.

Furthermore, when using object placement, hybrid memory systems using RRAM or STT-RAM are always more energy-efficient than a DRAM-only system. This is not the case for PCM, however. It is also not the case when using pages or blocks as the granularity for placing data.

PCM, with the parameters discussed in the literature, does not appear to be an energy-efficient main memory contender for the majority of benchmarks. It is effective only for benchmarks dominated by static energy. CJPEG shows up to 32.69 % energy savings under the DRAM-PCM configuration. For other benchmarks, PCM does not behave well due to the high dynamic energy and high latency. Page and block placement in this case performs worse than a DRAM-only system.

In the high-end scenario, dominated by static energy, we see little difference between the memory technologies (Table 7). This results from comparable parameters for static energy for the technologies (Table 1). Moreover, the NVM-only configuration incurs less slowdown than in the embedded scenario. As such, the energy saving of the NVM-only configuration remains positive.

6.6 Data Migration

So far we have focused on static data placement. Static data placement works very well for the majority of objects as they are typically accessed in a fairly uniform way. Moreover, application objects have a relatively short lifetime, on average less than 1 ms (Figure 4). As such, the opportunities for migration of objects are limited: by the time an object is migrated, it would already be deallocated. In contrast, virtual memory pages and blocks live throughout the life of the application and may store many different objects as they are allocated and freed. At this level, migration is mandatory as data access properties can vary widely.

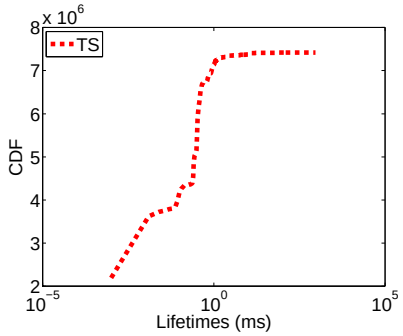


Figure 4: TS Benchmark: CDF of object lifetimes

Table 9: Mean lifetime per object in milliseconds.

Benchmark	Time	Benchmark	Time
473.astar	9.9	CJ	0.03
456.hmmer	0.30	LM	0.41
401.bzip2	5.47	SMG	93.33
458.sjeng	0.40	ESP	0.03
462.libquantum	0.04	GS	0.04
464.h264ref	0.49	X264	0.52
429.mcf	395	TS	0.05

Four benchmarks have object lifetimes significantly larger than 1 ms (Table 9). For these benchmarks, the argument of cost of migration in terms of both time and energy does not hold. Migration may be applicable to the long-living objects, but it makes sense only if (i) these objects attract many memory accesses and (ii) the memory accesses are concentrated to parts of the objects in any period of time.

7 Conclusion

This paper presents a software-only approach to reduce the energy consumption in hybrid DRAM/NVM memory that outperforms state-of-the-art hardware and OS page migration techniques. Key to our approach is to operate on application-level objects, which are program variables and memory allocation blocks. We demonstrate that the majority of objects have very stable properties for placement in hybrid memories and are too short-lived to warrant migration.

Object placement provides excellent energy savings except in cases where objects consist of multiple virtual memory pages *and* distinct pages exhibit distinct preference to DRAM or NVM. We consider that, in order to reduce the cost of online page migration techniques, it may be worthwhile to restrict online page migration to specific objects where this behavior occurs.

We have also investigated data placement at the very fine granularity of 64-byte blocks. While this is technically challenging to implement in real hardware, it does provide substantially higher energy savings than page placement. Again, limiting the technique to those objects where it may be beneficial is crucial to achieving an overall energy gain.

8 Acknowledgement

We thank our shepherd, Diana Franklin, and the anonymous reviewers for their valuable feedback. This work is supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the NovoSoft project (Marie Curie Actions, grant agreement 327744) and the NanoStreams project (grant agreement 610509).

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pp. 72–81, 2008.

[2] N. Binkert, *et al*. The GEM5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[3] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[4] G. Burr, *et al*. Overview of candidate device technologies for storage-class memory. *IBM J. of Res. and Dev.*, 52(4.5):449–464, July 2008.

[5] B. Calder, C. Krintz, J. Simmi and T. Austin. Cache-conscious data placement. *ASPLOS*, pp. 139–149, 1998.

[6] S. Cho, P.-C. Yew, and G. Lee. Decoupling local variable accesses in a wide-issue superscalar processor. In *ISCA*, pp. 100–110, 1999.

[7] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *DAC*, pp. 664–469, 2009.

[8] E. Doller. Forging a future in memory - new technologies, new markets, new applications. In *Hot Chips Tutorials*, 2010.

[9] X. Dong, *et al*. NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *TCAD*, 31(7):994–1007, July 2012.

[10] M. R. Guthaus, *et al*. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC*, pp. 3–14, 2001.

[11] Y. Ho, G. M. Huang, and P. Li. Nonvolatile memristor memory: Device characteristics and design implications. In *ICCAD*, pp. 485–490, 2009.

[12] J. Hu, *et al*. Data allocation optimization for hybrid scratch pad memory with sram and nonvolatile memory. *TVLSI*, 21(6):1094–1102, June 2013.

[13] C.-C. Huang and V. Nagarajan. ATCache: Reducing DRAM cache latency via a small SRAM tag cache. In *PACT*, pp. 51–60, 2014.

[14] E. Kultursay, *et al*. Evaluating STT-RAM as an energy-efficient main memory alternative. In *ISPASS*, pp. 256–267, April 2013.

[15] B. C. Lee, *et al*. Architecting phase change memory as a scalable dram alternative. In *ISCA*, pp. 2–13, 2009.

[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, pages 330–335, Dec. 1997.

[17] H.-H. S. Lee and G. S. Tyson. Region-based caching: An energy-delay efficient memory architecture for embedded processors. In *CASES*, pp. 120–127, 2000.

[18] D. Li, *et al*. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *IPDPS*, pp. 945–956, 2012.

[19] Q. Li, *et al*. Compiler-assisted preferred caching for embedded systems with STT-RAM based hybrid cache. In *LCTES*, pp. 109–118, 2012.

[20] The LLVM compiler infrastructure. http://llvm.org.

[21] J. Meza, J. Li, and O. Mutlu. Evaluating row buffer locality in future non-volatile main memories. SAFARI Technical Report No. 2012-002, Carnegie Mellon University, Dec. 2012.

[22] Micron TN-41-01: Calculating memory system power. http://www.micron.com/products/support/power-calc. Last consulted: Nov 2014.

[23] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *HotOS*, pp. 14–14, 2009.

[24] T. Perez and C. De Rosa. Non-volatile memory: Emerging technologies and their impacts on memory systems. Technical report, 2010.

[25] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA*, pp. 24–33, 2009.

[26] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *ICS*, pp. 85–95, 2011.

[27] S. Raoux, *et al*. Phase-change random access memory: A scalable technology. *IBM J. of Res. and Dev.*, 52(4.5):465–479, July 2008.

[28] S. Rul, H. Vandierendonck, and K. De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Comput.*, 36(9):531–551, Sept. 2010.

[29] SPEC Benchmark Suite https://www.spec.org/cpu.

[30] H. Vandierendonck, A. Hassan, and D. Nikolopoulos. On the energy-efficiency of byte-addressable non-volatile memory. *Comput. Archit. Letters*, PP(99):1–1, 2014.

[31] Z. Wang, Z. Gu, and Z. Shao. Optimizated allocation of data variables to PCM/DRAM-based hybrid main memory for real-time embedded systems. *Embedded Systems Letters, IEEE*, 6(3):61–64, Sept 2014.

[32] H. Yoon. Row buffer locality aware caching policies for hybrid memories. In *ICCD*, pp. 337–344, 2012.

[33] O. Zilberberg, S. Weiss, and S. Toledo. Phase-change memory: An architectural perspective. *ACM Comput. Surv.*, 45(3):29:1–29:33, July 2013.